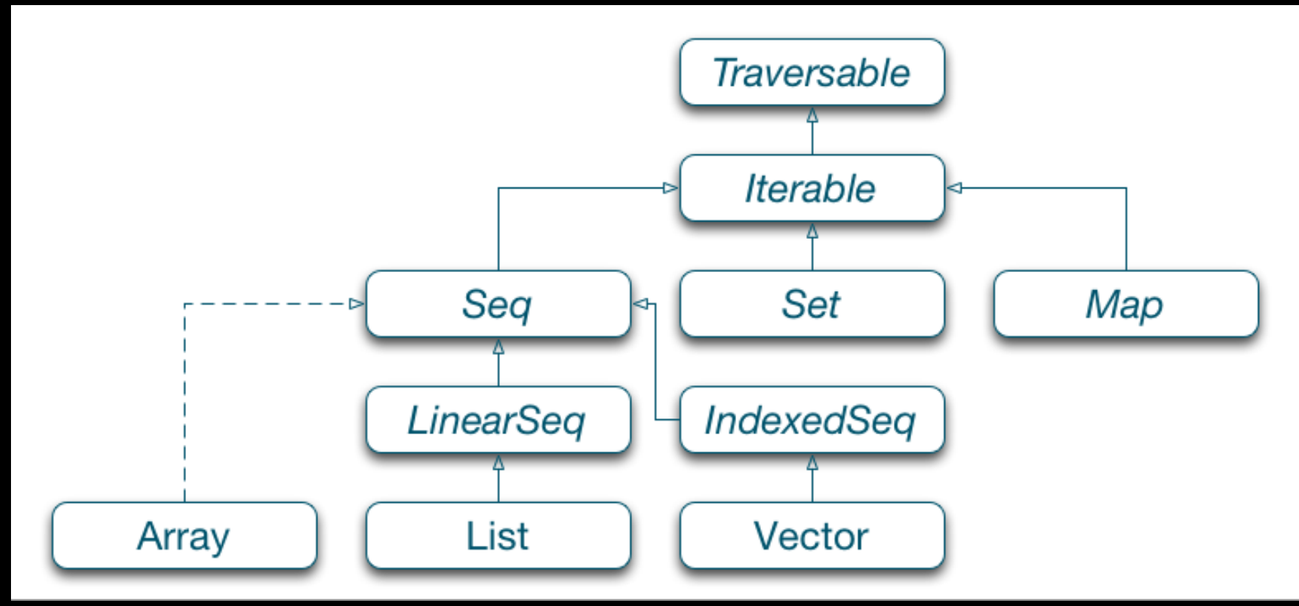
Scala for Professionals

Custom Scala Collections

Recap: The Scala Collection Hierarchy

- The Scala Collection library is *very* comprehensive.
- Some of the more important collection types are outlined on the sidebar.
- We may note that many of the 'base' collection methods are defined in `Traversable`.



Uniform Return Type Principle

Common methods – such as `filter`, `take`, and in many cases `map` and `flatMap` – preserve the starting collection type:*

```
Vector(1, 2, 3) take 2
// res0: Vector[Int] = Vector(1, 2)

Seq(1, 2, 3) take 2
// res1: Seq[Int] = List(1, 2)

Traversable(1, 2, 3) take 2
// res2: Traversable[Int] = List(1, 2)
```

- How could we implement this **uniform return type principle** in our own code?

---

* Though the inner type (the type parameter) can change, of course.

Uniform Return Type Principle

Implementing it the Cumbersome Way

Through Scala 2.7, the unifrom return type principle was implemented through code duplication.

```
trait Traversable[A] {
  def filter(p: A => Boolean): Traversable[A] =
  // ...
}

trait Iterable[A] extends Traversable[A] {
  override def filter(p: A => Boolean): Iterable[A] =
  // ...
}
```

- Code duplication is cumbersome: Implementations are often identical, and refactoring can be dangerous.
- Specifically, code duplication leads to ‘bit rot’ – inconsistencies, broken window effects, etc.

Uniform Return Type Principle

The Clever Approach

In Scala 2.8, the collection library was rewritten.

- Now, the uniform return type principle is implemented by abstracting over return types:
  - In most cases, it abstracts over return types with ‘Builders’, e.g. with `filter` and `take`
  - In trickier cases, it abstracts over return types with Type Classes, e.g. with `map` and `flatMap`

Collection Builders

Builders Can Support Most Cases

For many collection methods, the return type can be abstracted out by introducing *collection specific builders*. A `Builder` is a mutable data structure, such as `ListBuffer`.

Here are the core methods on the `Builder` trait:

```
trait Builder[-Elem, +To] {
  def +=(elem: Elem): this.type
  def clear()
  def result(): To
  // ...
}
```

A `Builder` is not meant as a direct collection itself, rather, it returns a final value representing the 'built' collection. Its return type is abstracted out by the `To` parameter, returned in `result`

`Like` Traits

Most collection methods are defined in `Like`-traits, e.g. `SeqLike`. Collection implementations mix-in these `Like`-traits to provide appropriate functionality:

```
trait Seq[+A] extends /*...*/ with SeqLike[A, Seq[A]]
```

- The `Like`-traits are parameterized with the element type and the collection they're mixed into.

Builders In Action

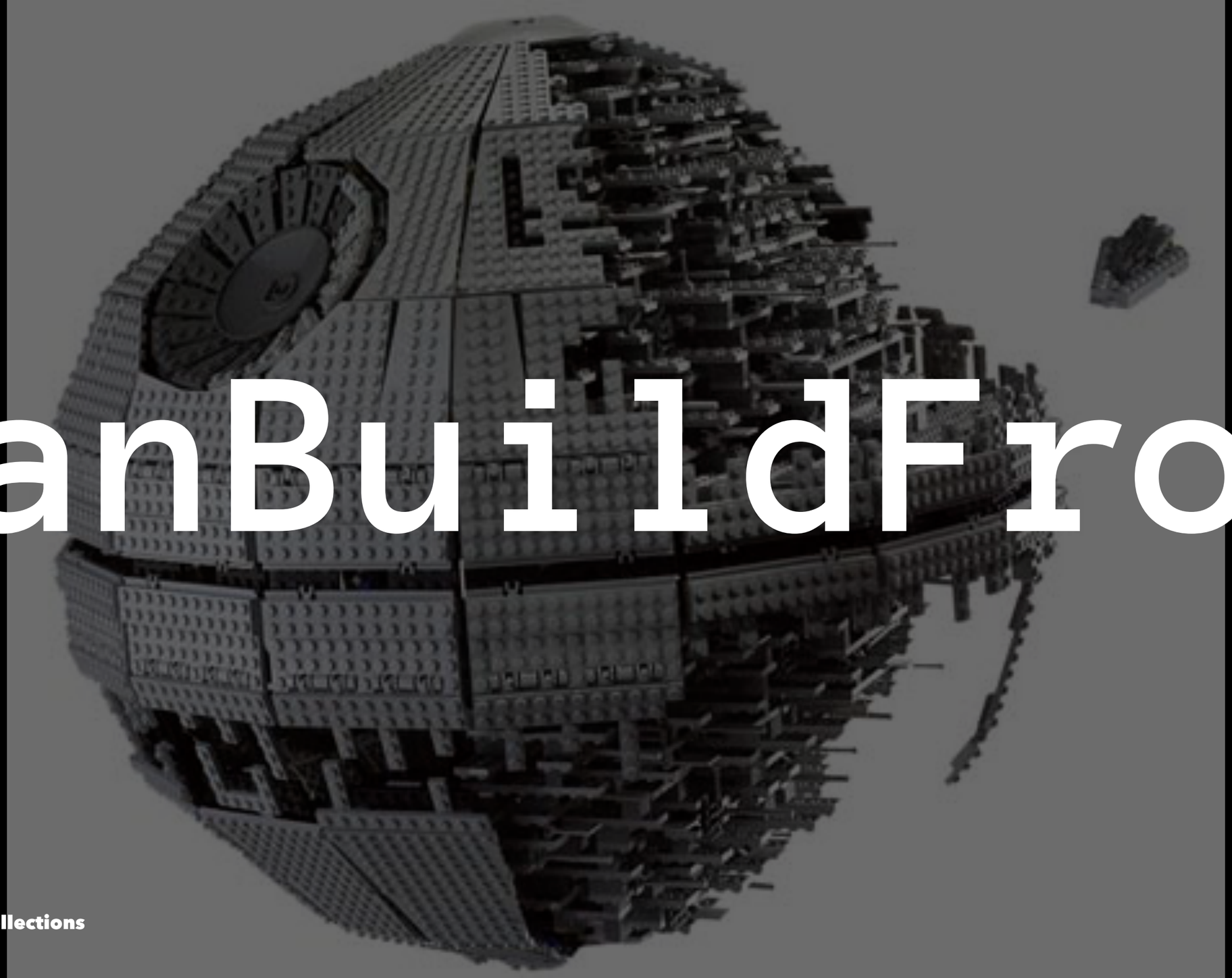
By Example

Many methods are defined in terms of the `newBuilder` method:

```
trait TraversableLike[+A, +Repr] {
  def filter(p: A => Boolean): Repr = {
    val b = newBuilder
    for (x <- this)
      if (p(x)) b += x
    b.result
  }

  def newBuilder: Builder[A, Repr]
```

- The return type is abstracted out by the `Repr` type parameter.
- By mixing a `Like`-trait into a collection and defining the `newBuilder` method, the collection defines the return type.



CanBuildFrom

Uniform Return Type Principle

When It Breaks

For methods which can change their element type, the collection type might have to be changed:

```
"abc" map (_ + 1)
// res0: IndexedSeq[Int] = Vector(98, 99, 100)

BitSet(1, 2, 3) map (_ * 0.5)
// res1: SortedSet[Double] = TreeSet(0.5, 1.0, 1.5)
```

- However, sometimes the uniform return type principles *does* hold:

```
BitSet(1, 2, 3) map (_ * 2)

// res0: BitSet = BitSet(2, 4, 6)
```

- **Quiz**: Why?

Type Classes for the Tricky Cases

The issue at hand really boils down to the following question:
 - For a given collection type `From` and a given element type `Elem`, which collection types `To` can be constructed?
 - This is encoded in Scala's type class `CanBuildFrom`:

```scala
trait CanBuildFrom[-From, -Elem, +To] {
  def apply(): Builder[Elem, To]
  // ...
}
```

- `CanBuildFrom` is a factory for `Builders`.

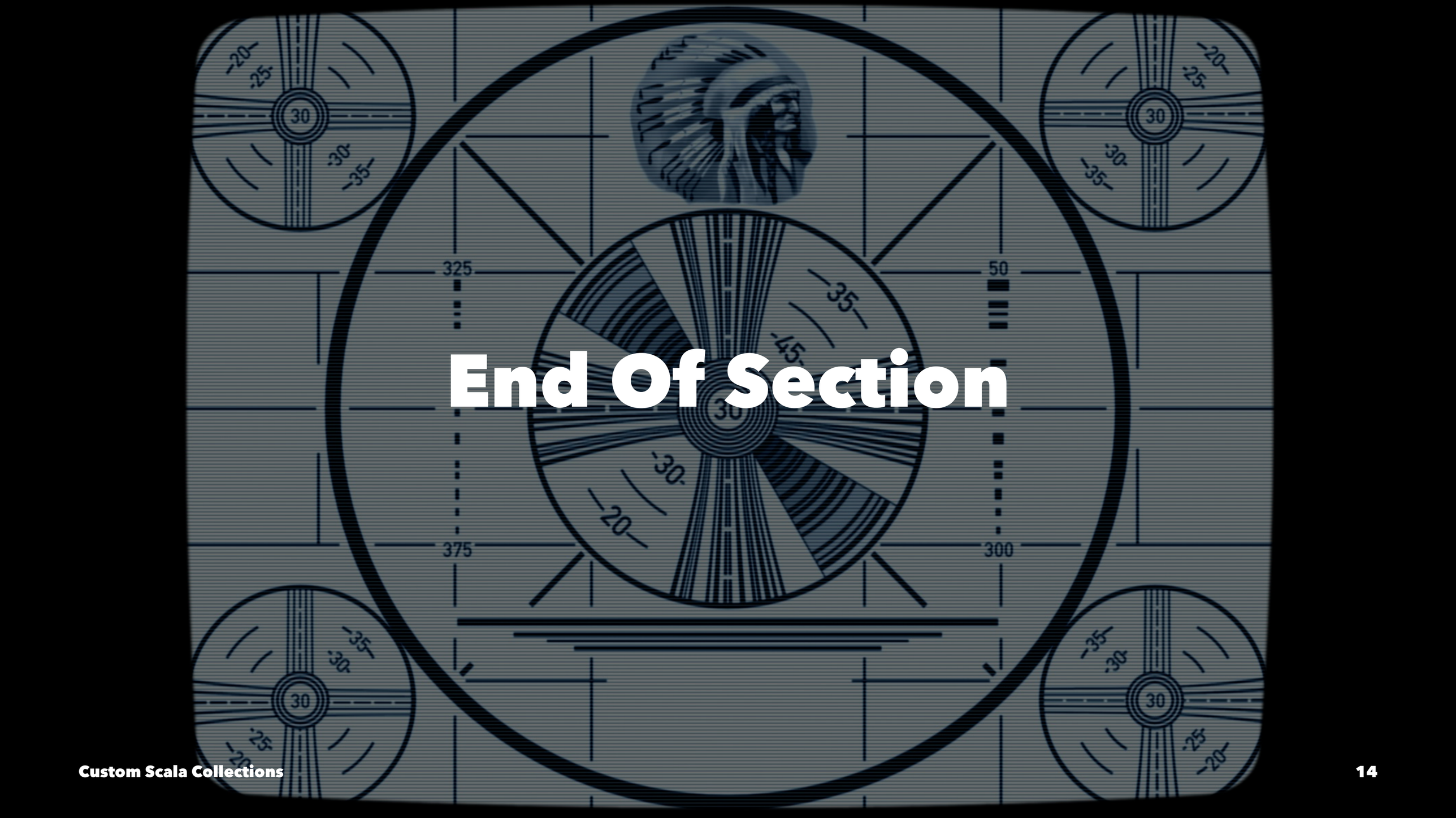
Example: `CanBuildFrom` in Action

Methods which can change their element type are defined with an implicit `CanBuildFrom` parameter:

```
trait TraversableLike[+A, +Repr] {
  def map[B, That](f: A => B)(
      implicit bf: CanBuildFrom[Repr, B, That]): That = {
    val b = bf(repr)
    for (x <- this) b += f(x)
    b.result
  }
  // ...
}
```

- Example for a `CanBuildFrom` type class instance:

```
implicit def canBuildFrom: CanBuildFrom[BitSet, Int, BitSet] = // ...
```


End Of Section